GREEDY TECHNIQUE

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step and this is the central point of this technique.

The choice made must be:

- *feasible*, i.e., it has to satisfy the problem'sconstraints
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of thealgorithm

Greedy Technique algorithms are:

- Prim'salgorithm
- Kruskal'sAlgorithm
- Dijkstra'sAlgorithm
- HuffmanTrees

Two classic algorithms for the minimum spanning tree problem: Prim's algorithm and Kruskal's algorithm. They solve the same problem by applying the greedy approach in two different ways, and both of them always yield an optimalsolution.

Another classic algorithm named Dijkstra's algorithm used to find the shortest-path in a weighted graph problem solved by Greedy Technique . Huffman codes is an important data compression method that can be interpreted as an application of the greedy technique.

The first way is one of the common ways to do the proof for Greedy Technique is by

mathematical induction.

The second way to prove optimality of a greedy algorithm is to show that on each step it does at least as well as any other algorithm could in **advancing** toward the problem'sgoal.

Example: find the minimum number of moves needed for a chess knight to go from one corner of a 100 × 100 board to the diagonally opposite corner. (The knight's moves are L-shaped jumps: two squares horizontally or vertically followed by one square in the perpendicular direction.)

A greedy solution is clear here: jump as close to the goal as possible on each move. Thus, if its start and finish squares are (1,1) and (100, 100), respectively, a sequence of 66 moves such as (1, 1) − (3, 2) − (4, 4) − . . . − (97, 97) − (99, 98) − (100, 100) solves the problem(The number k of two-move advances can be obtained from the equation 1+ 3k =100).

Why is this a minimum-move solution? Because if we measure the distance to the goal by the Manhattan distance, which is the sum of the difference between the row numbers and the difference between the column numbers of two squares in question, the greedy algorithm decreases it by 3 on each move.

The third way is simply to show that the final result obtained by a greedy algorithm is optimal based on the **algorithm's output** rather than the way it operates.

Example: Consider the problem of placing the maximum number of chips on an 8 × 8 board so that no two chips are placed on the same or adjacent vertically, horizontally, or diagonally.

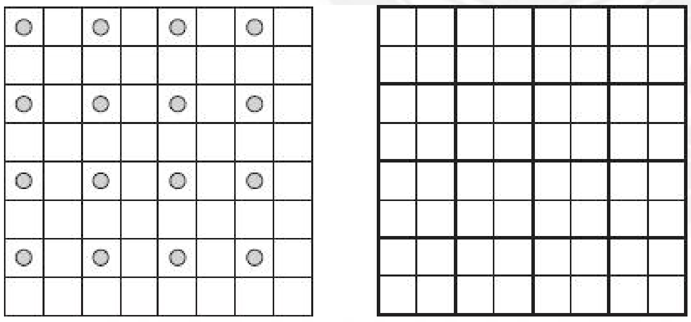


FIGURE 3.12 (a) Placement of 16 chips on non-adjacent squares. (b) Partition of the board proving impossibility of placing more than 16chips.

It is impossible to place more than one chip in each of these squares, which implies that the total number of nonadjacent chips on the board cannot exceed 16.

PRIM'SALGORITHM

A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges. The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.

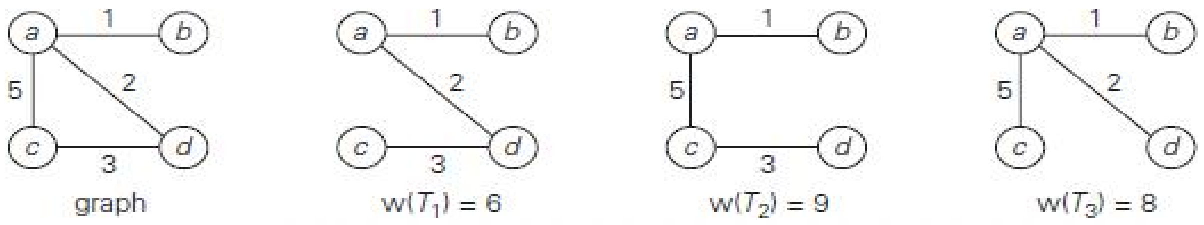


FIGURE 3.13 Graph and its spanning trees, with *T*1 being the minimum spanning tree.

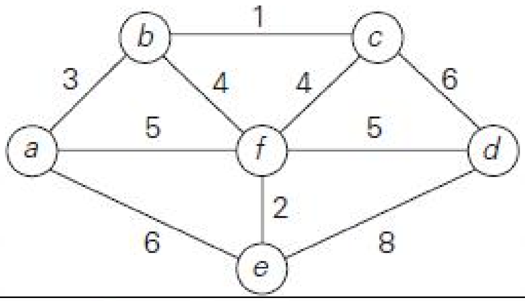
The minimum spanning tree is illustrated in Figure 3. If we were to try constructing a minimum spanning tree by exhaustive search, we would face two serious obstacles. First, the number of spanning trees grows exponentially with the graph size (at least for dense graphs). Second, generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a minimum spanning tree for a weighted graph

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

ALGORITHM *Prim(G)*

```
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \{V, E\}$
//Output: $E_T$, the set of edges composing a minimum
spanning tree of $G$ $V_T \leftarrow \{v_0\}$ //the set of tree vertices can be
initialized with any vertex $E_T \leftarrow \Phi$
for $i \leftarrow 1$ to $|V| - 1$ do
    findaminimum-
    weightedgee□=(v□,u□)amongalltheedges(v,u) such that $v$
    is in $V_T$ and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return $E_T$
```

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is O(|E| log |V |) in a connected graph, where |V| − 1≤ |E|.



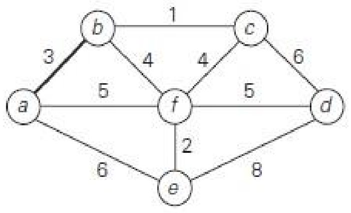
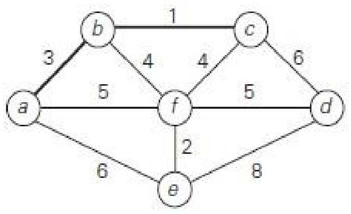
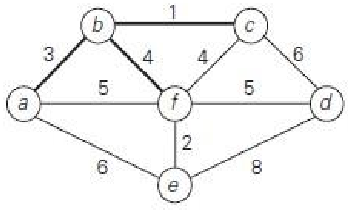
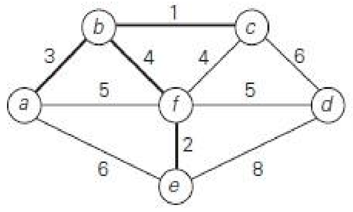
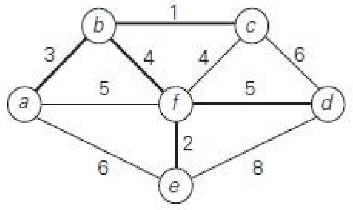
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

FIGURE 3.14 Application of Prim's algorithm. The parenthesized labels of a vertex in the middle
column indicate the nearest tree vertex and edge weight; selected vertices and edges are in bold.

KRUSKAL'SALGORITHM

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph G= {V, E} as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest. the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph G = (V, E) as an acyclic subgraph with |V| − 1 edges for which the sum of the edge weights is the smallest.

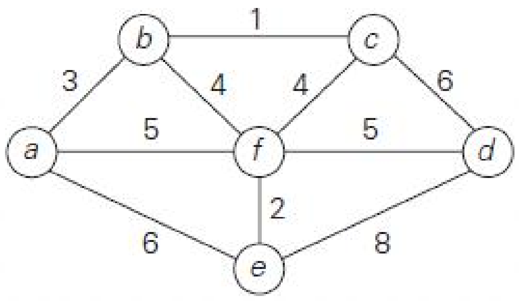
ALGORITHM *Kruskal(G)*

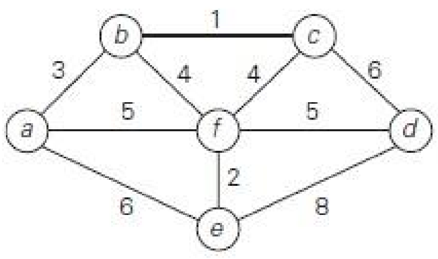
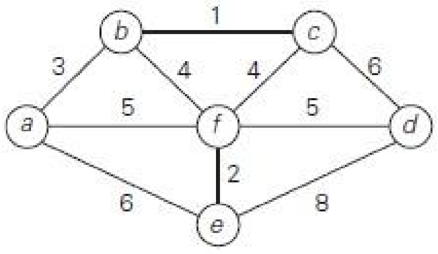
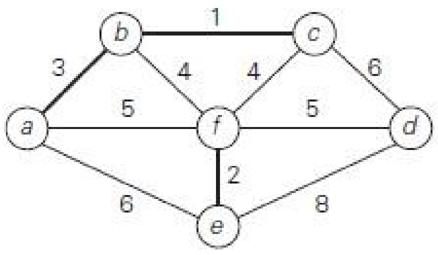
```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = (V, E)$
//Output: $E_T$, the set of edges composing a minimum spanning tree of $G$
        sort $E$ in nondecreasing order of the edge weights $w(e_{i1}) \le .$
. . $\le w(e_{i|E|})$ $E_T \leftarrow \Phi$; $ecounter \leftarrow 0$  //initialize the set of tree edges and
itssize

$K \leftarrow 0$                    //initialize the number of processededges
while $ecounter < |V| - 1$ do
      $k \leftarrow k + 1$
      if $E_T \cup \{e_{ik}\}$ is acyclic
            $E_T \leftarrow E_T \cup \{e_{ik}\}$; $ecounter \leftarrow ecounter + 1$
return $E_T$
```

The initial forest consists of |V | trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).

Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called union-find algorithms. With an efficient union-find algorithm, the running time of Kruskal’s algorithm will be $O(|E| \log |E|)$.



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

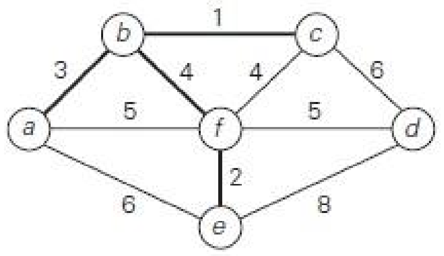
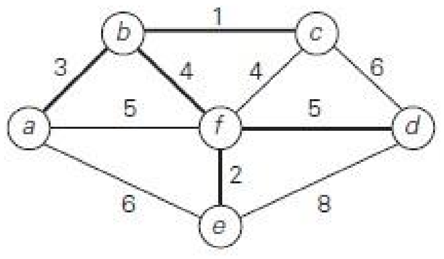
ab 3	bc 1	ef 2	ab 3	**bf** **4**	cf 4	af 5	df 5	ae 6	cd 6	de 8	
bf 4	bc 1	ef 2	ab 3	bf 4	cf 4	af 5	**df** **5**	ae 6	cd 6	de 8	
df 5											

FIGURE 3.15 Application of Kruskal's algorithm. Selected edges are shown in bold.DIJKSTRA'S ALGORITHM

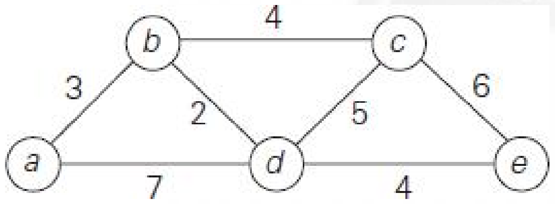
- Dijkstra's Algorithm solves the **single-source shortest-pathsproblem**.
- For a given vertex called the ***source*** in a weighted connected graph, find shortest paths to all its othervertices.
- The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have **edges in common**.
- The most widely used **applications** are transportation planning and packet routing in communication networks including the Internet.
- It also includes **finding shortest paths** in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling.
- In the world of **entertainment**, one can mention pathfinding in video games and finding best solutions to puzzles using their state-spacegraphs.
- Dijkstra's algorithm is the best-known algorithm for the single-source shortest-paths problem.

ALGORITHM *Dijkstra(G,s)*

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = (V, E) with nonnegative weights and its
vertex s
//Output: The length dv of a shortest path from s to v and its penultimate vertex pv for
every
//          vertex v in V
```

```
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
        $dv \leftarrow \infty$; $pv \leftarrow$ null
        Insert (Q, v, dv) //initialize vertex priority in the priority queue
$Ds \leftarrow 0$; Decrease(Q, s, $d_s$) //update priority
of s with $d_s$ $V_T \leftarrow \Phi$
for $i \leftarrow 0$ to $|V| - 1$ do
        $u^* \leftarrow$ DeleteMin(Q) //delete the minimum priority element
        $V_T \leftarrow V_T \cup \{u^*\}$
        for every vertex u in V − VT that is
                adjacent to $u^*$ do if $d_u^* + w(u^*, u) < d_u$
                $d_u \leftarrow d_u^* + w(u^*, u)$;
                $p_u \leftarrow u^*$ Decrease(Q,
                u, $d_u$)
```

The time efficiency of Dijkstra’s algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. It is in $\Theta(|V|^2)$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency lists and the priority queue implemented as a min- heap, it is in $O(|E| \log |V|)$.



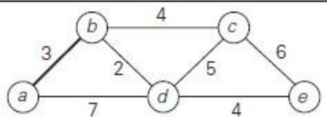
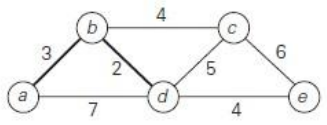
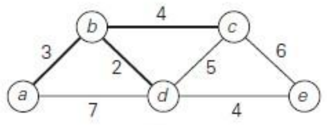
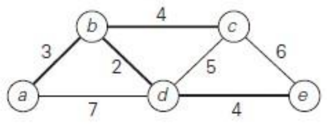
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

FIGURE 3.16 Application of Dijkstra's algorithm. The next closest vertex is shown in bold

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

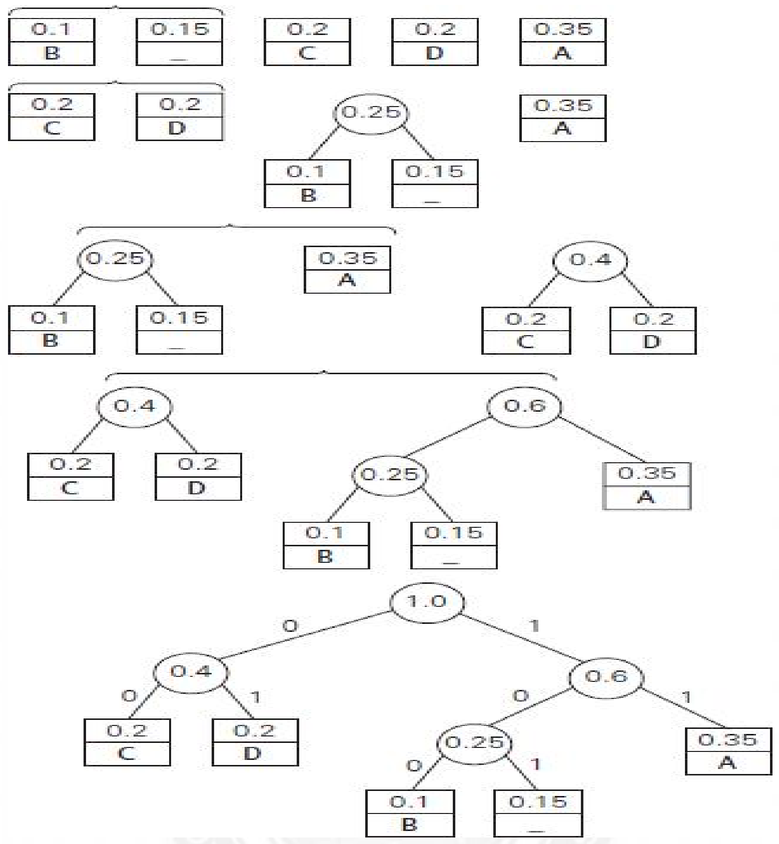
From a to b : a − b of
length 3 From a to d : a
− b − d of length 5
From a to c : a − b − c
of length 7
From a to e : a − b − d − e of length 9

HUFFMANTREES

To encode a text that comprises symbols from some *n*-symbol alphabet by assigning to each of the text's symbols some sequence of bits called the ***codeword***. For example, we can use a ***fixed- length encoding*** that assigns to each symbol a bit string of the same length *m* (*m* ≥ log2 *n*). This is exactly what the standard ASCII code does.

Variable-length encoding, which assigns codewords of different lengths to different symbols, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the *i*th) symbol? To avoid this complication, we can limit urselvesto the so-called ***prefix-free*** (or simply ***prefix***) ***codes***.

In a prefix code, no codeword is a prefix of a codeword of another symbol. Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some symbol, replace these bits by this symbol, and repeat this operation until the bit string's end isreached.



Huffman's algorithm

Step 1 Initialize *n* one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as itsweight.

A tree constructed by the above algorithm is called a **Huffman tree**. It defines in the manner described above is called a **Huffman code**.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 3.18

FIGURE 3.18 Example of constructing a Huffman coding tree.

The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD. With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is 2 .0.35 + 3 .0.1+ 2 .0.2 + 2 .0.2 + 3 .0.15 =2.25.

We used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. Thus, for this toy example, Huffman's code achieves the ***compression ratio* -** a standard measure of a compression algorithm's effectiveness of (3−2.25) / 3 · 100% = **25%**. In other words, Huffman's encoding of the text will use 25% less memory than its fixed-length encoding.

Running time is O(*n log n*), as each priority queue operation takes time O(*log n*).

Applications of Huffman's encoding

1. Huffman's encoding is a variable length encoding, so that number of bits used are lesser than fixed lengthencoding.
2. Huffman's encoding is very useful for filecompression.
3. Huffman's code is used in transmission of data in an encodedformat.
4. Huffman's encoding is used in decision trees and gameplaying.